Cypress Semiconductor
ARM Cortex-M3 PSoC 5 Design Challenge

Implementing FULL CAN bus in PSoC5 for electric vehicle development

Step by Step Instructions

By
M.Sc. Hakam Saffour

Supervised by
Prof. Dr.-Ing. Roland Kasper
Otto Von Guericke University Magdeburg
Faculty of Mechanical Engineering
Institute of Mobile systems- Mechatronics

April 2011

1. Introduction

This report describes the steps required to implement and program PSoC5 system-on-chip microcontroller to utilize CAN controller. Basic information about CAN bus theory will be presented, and then step by step of implementing CAN bus is described.

It is expected from the reader to have good knowledge of CAN theory in addition to good experience with using PSoC Creator.

The objective is to let two PSoC5 controllers and another third controller communicate with each other utilizing CAN bus, including the hardware and software setup.

2. CAN Bus

The Controller Area Network (CAN) specification defines the Data Link Layer, ISO 11898 defines the Physical Layer.

The CAN bus is a Balanced (differential) 2-wire interface running over either a Shielded Twisted Pair (STP), Un-shielded Twisted Pair (UTP), or Ribbon cable. Each node uses a Male 9-pin D connector.

2.1. Bit Encoding:

The Bit Encoding used is: Non Return to Zero (NRZ) encoding (with bit-stuffing) for data communication on a differential two wire bus. The use of NRZ encoding ensures compact messages with a minimum number of transitions and high resilience to external disturbance.

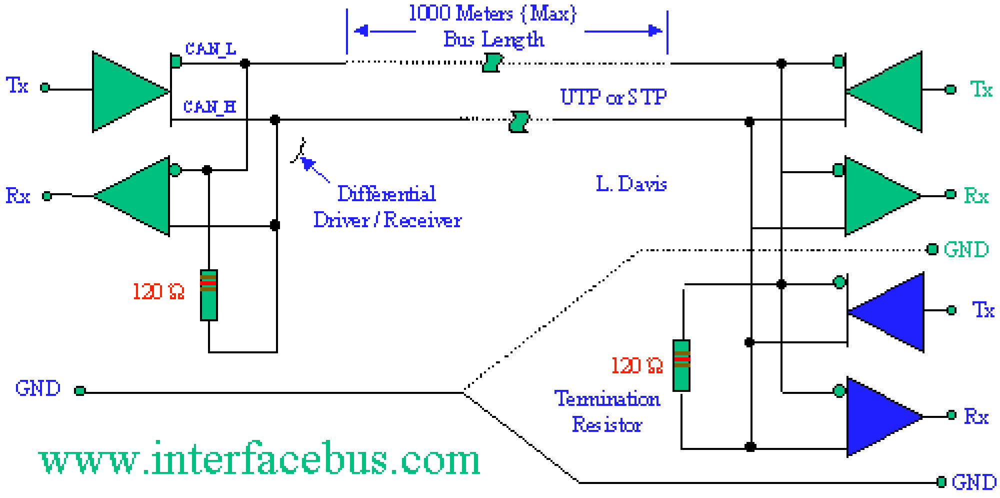

Figure 1: CAN Bus Electrical Interface Circuit

2.2. Data Rate:

A number of different data rates are defined, with 1Mbps (Bits per second) being the top end, and 10kbps the minimum rate.

2.3. Cable Length:

Cable length depends on the data rate used. The maximum line length is 1Km (40 meters at 1Mbps). Termination resistors are used at each end of the cable. The worst-case transmission time of an 8-byte frame with an 11-bit identifier is 134 bit times (that's 134 microseconds at the maximum baud rate of 1Mbits/sec).

2.4. CAN Message Frame:

The CAN Bus interface uses an asynchronous transmission scheme controlled by start and stop bits at the beginning and end of each character. This interface is used, employing serial binary interchange. Information is passed from transmitters to receivers in a data frame. The data frame is composed of an Arbitration field, Control field, Data field, CRC field, ACK field. The frame begins with a 'Start of frame' [SOF], and ends with an 'End of frame' [EOF] space. The data field may be from 0 to 8 bytes.

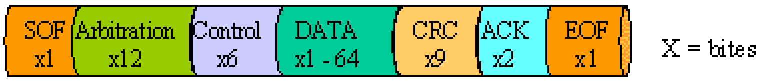

Figure 2: Can Message Frame

CAN implements five error detection mechanisms; 3 at the message level (by the receiver) and 2 at the bit level (by the transmitter) [Also incorporates error flags]. At the message level: Cyclic Redundancy Checks (CRC), Frame Checks, and Acknowledgment Error Checks. At the bit level: Bit Monitoring and Bit Stuffing.

2.5. CAN Transceiver:

There are different transceivers available in the market; in this report SN65HVD251 transceiver from Texas Instrument was used (please refer to the data sheet for more details). High input impedance of SN65HVD251 allows up to 120 nodes on a bus, also the unpowered node does not disturb the bus.

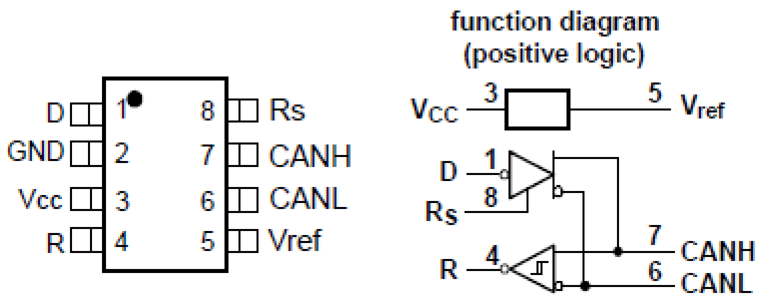

Figure 3: SN65HVD251 transceiver pin layout

The standard specifies the interconnection between CAN transceivers to be a single twisted-pair (shielded or unshielded) with 120 Ω resistor at both ends of the CAN bus cable. Figure 4 shows the typical hardware layout for CAN bus.

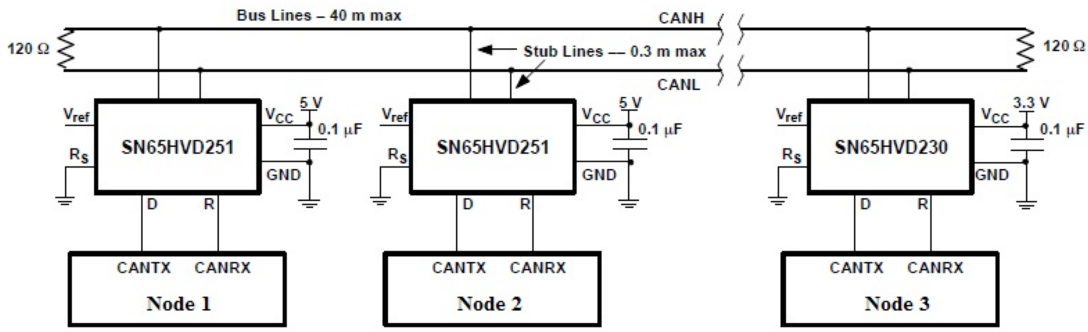

Figure 4: Typical CAN bus hardware connection

3. Hardware Setup

The below figure illustrate the complete hardware set up for three nodes (2x PSoC5 system-on-chip microcontrollers and mbed microcontroller). The development kits for the two PSoC5 are: CY8CKIT-014 PSoC5 First Touch Starter Kit and CY8CKIT-001 PSoC Development Kit

Note that node 2 is powered by 3.3VDC and node 1 & 3 is powered by 5VDC, however this does not affect the functionality of the CAN bus.

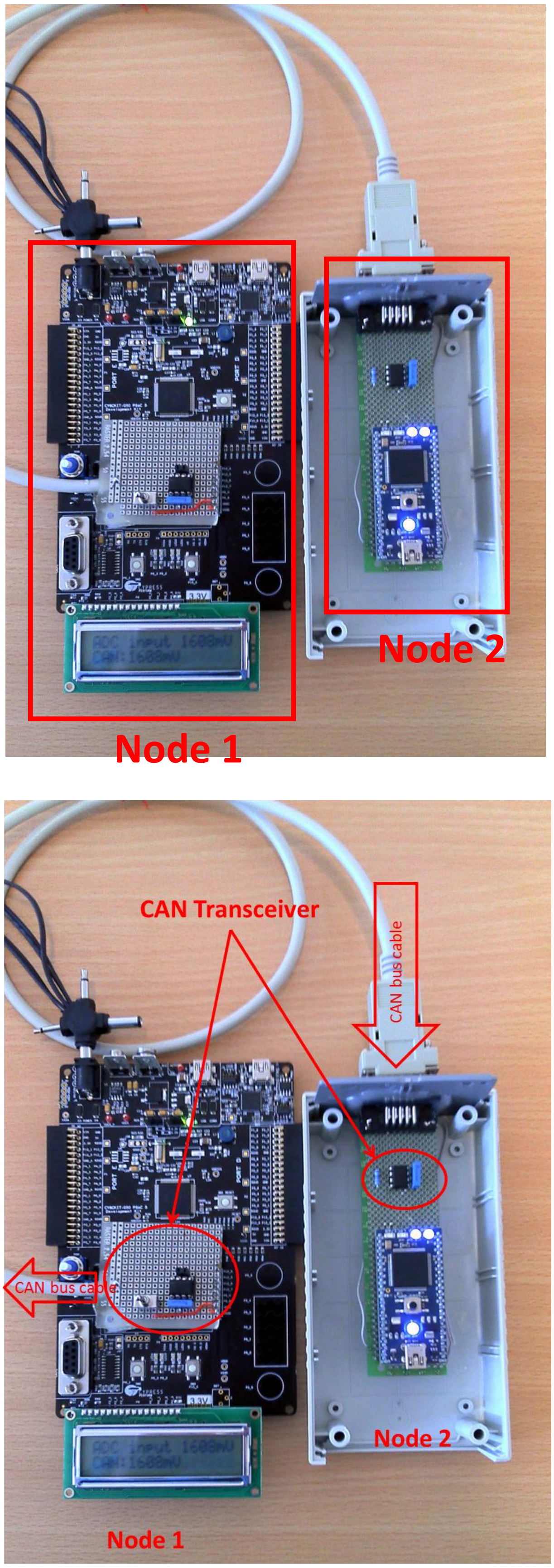

Figure 5: Complete hardware setup

4. Software Setup – Node 1

4.1. Project Parameters Setting

PSoc Creator 1.0 Beta 5.0 is used for developing the system. Simple flowchart for test purpose is shown in Figure 6 below.

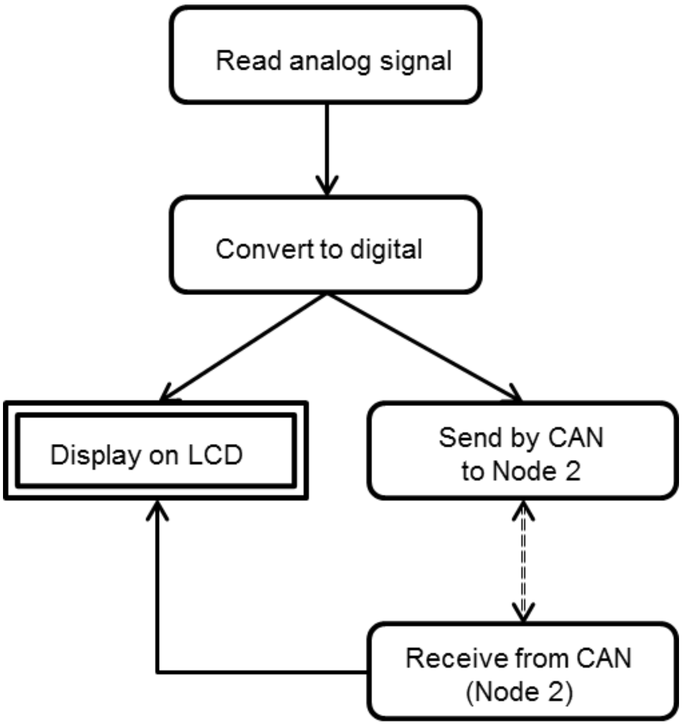

Figure 6: Flowchart

Note: *please refer to the data sheet of CAN controller in PSoC creator and instruction manual provided by Cypress website:*
(http://www.cypress.com/?rID=37766).

In the following pages the steps of setting up and programming PSoc5 to implement FULL CAN bus communication is described mostly through figures:

1- Make sure to select PSoC 5 option when creating new project (avoid using special characters like (&) in the directory name where the project files are saved).

2- Three components are used in the hardware design SAR ADC, LCD, and CAN controller.

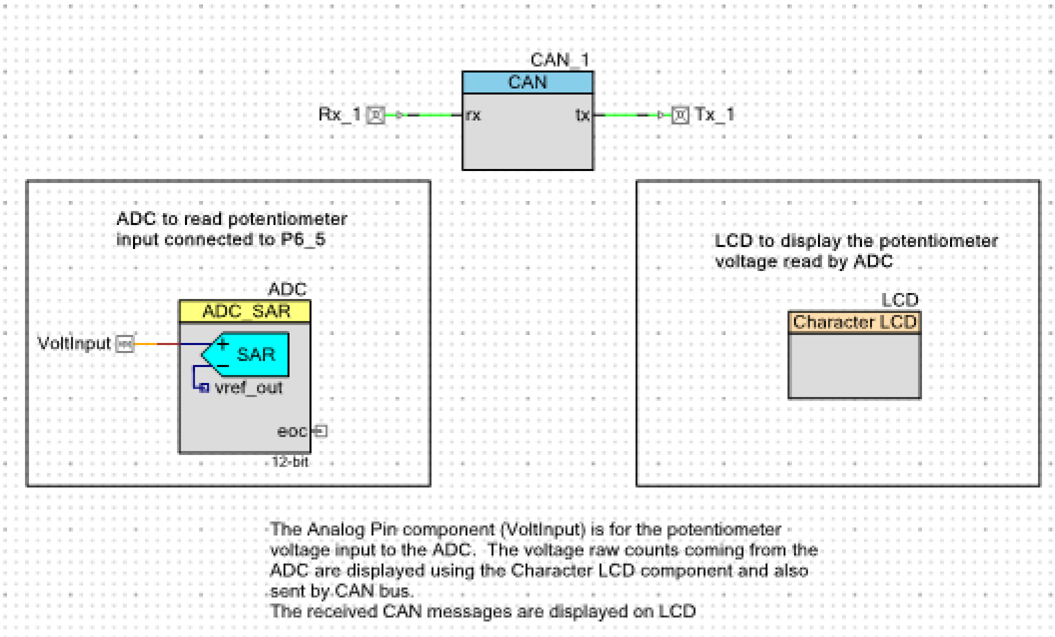

Figure 7: Top design layout

3- CAN Controller setting: leave the default settings in the “General” tab. In this example baud rate of 1000kbps is selected.

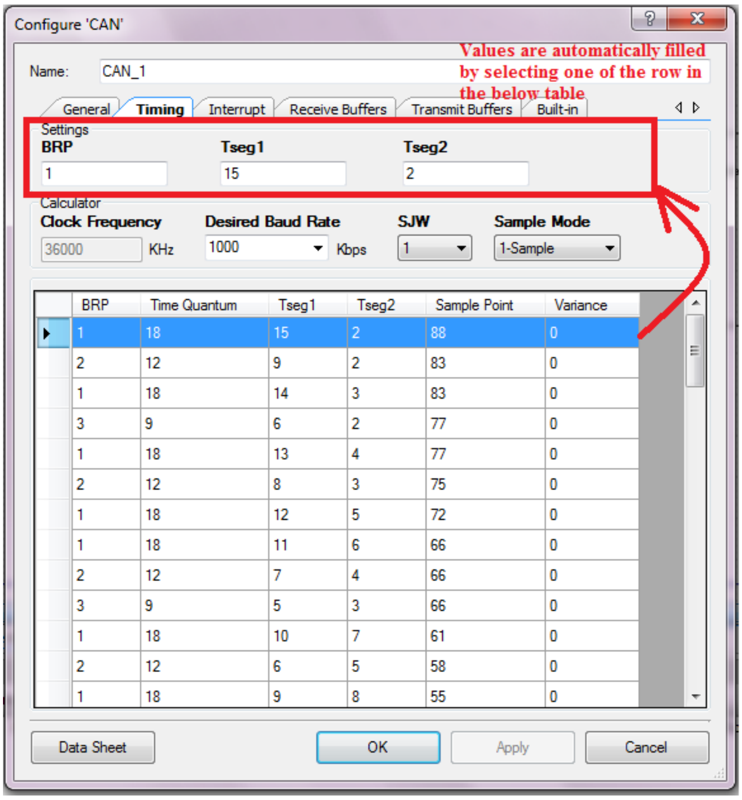

Figure 8: CAN controller configuration- timing

4- Leave the default settings in the “Interrupt” tab.

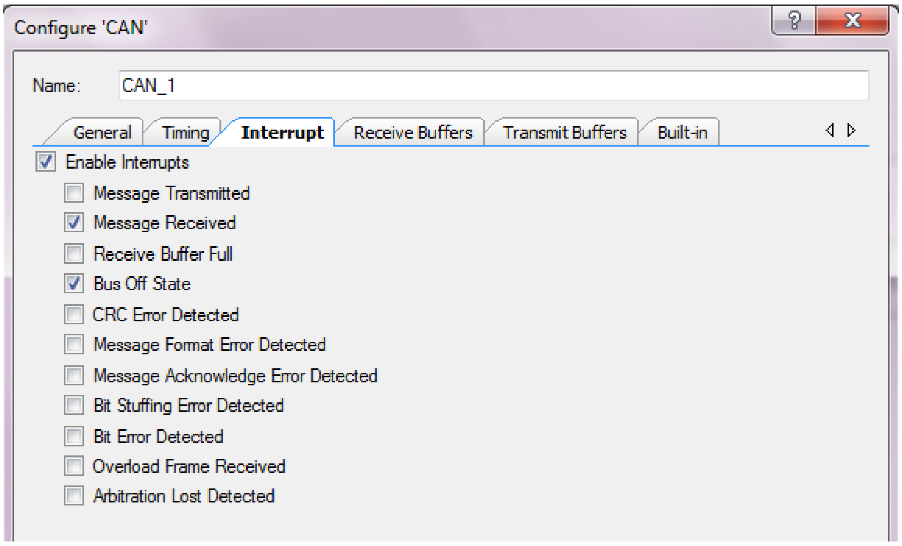

Figure 9: CAN controller configuration- Interrupt

5- The ID of the received message has to be specified and has to be the same ID of the transmitting node. Mailbox 0 has ID of 0x200 which is the same ID of Node 2 (the transmitter in this case). Note that the smaller the ID value the higher priority the message is.

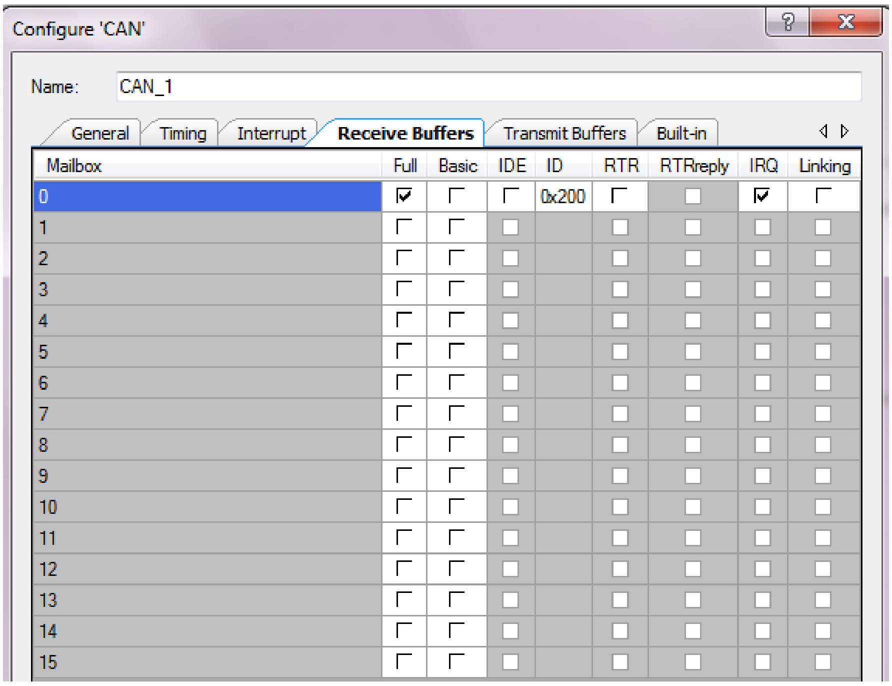

Figure 10: CAN controller configuration- Receive Buffers

6- The extended ID (29 bits) is used for the transmitted message. The message will be received by node 2, then node 2 will transmit the same message back.

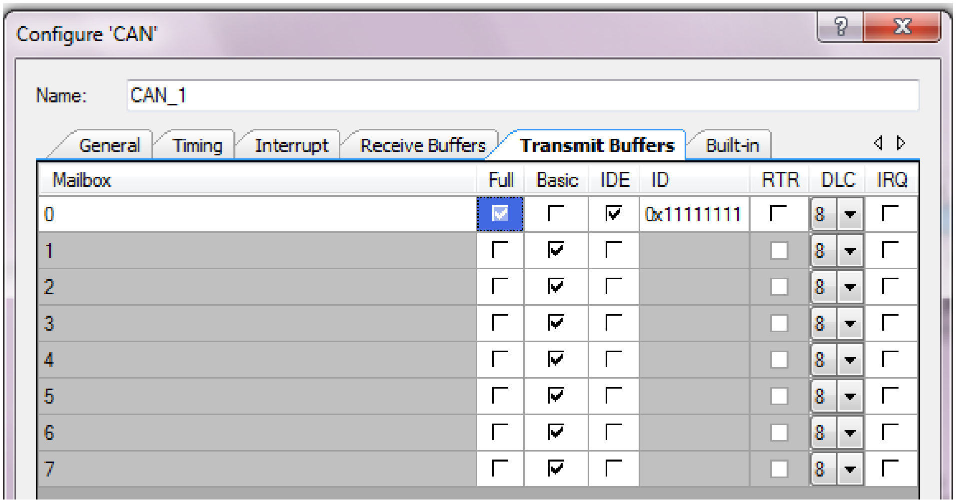

Figure 11: CAN controller configuration- Transmit Buffers

7- Leave the default settings for the LCD block.

8- 12 bit resolution for the SAR ADC is selected.

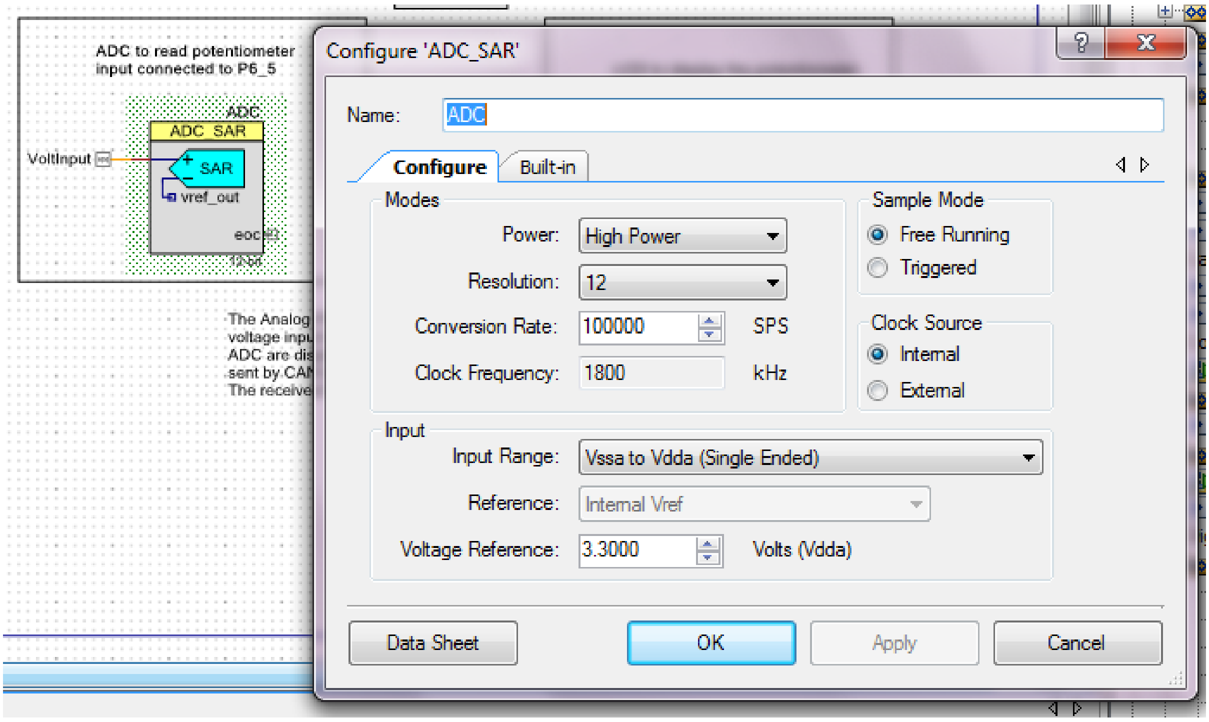

Figure 12: Delta sigma ADC configuration

9- Pin assignment of the controller can be defined as desired. Note that for CY8CKIT-050 PSoC Development Kit the LCD is connected to Port 2 [6:0].

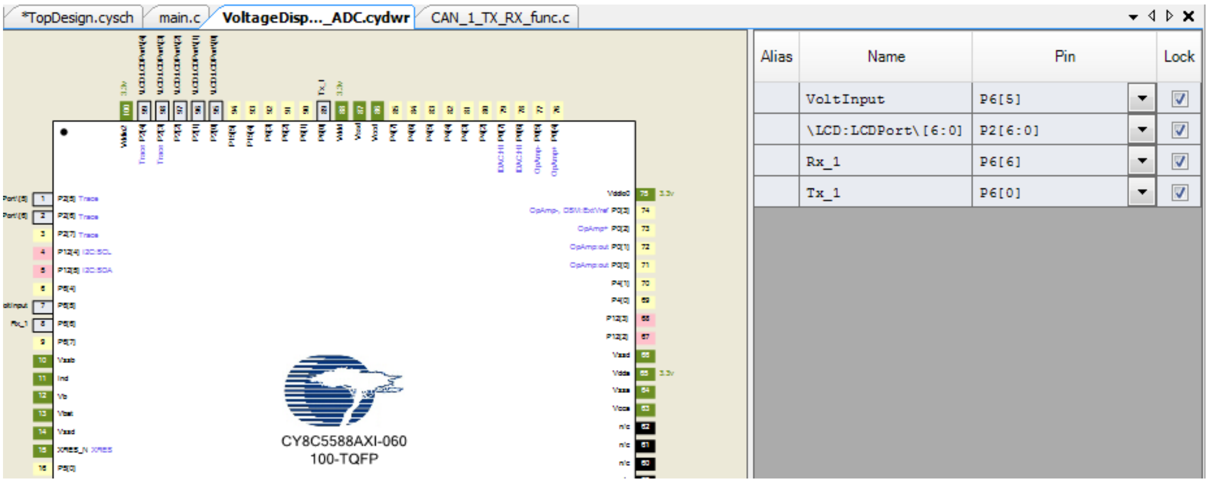

Figure 13: Pin assignment layout

4.2. C Code

Before writing the code in the “main.c” file, just build the project (shift+F6) in order to generate the associated files to the components selected in Figure 7.

1- The C code will mainly be written in the “main.c” file, however some additions are written to one of the generated files (/Generated_Source /PSoC5/CAN_1/CAN_1_TX_RX_func.c).

2- The message is sent by using “CAN_1_SendMsgx();”, while in case of receiving a message, the receive function “CAN_1_ReceiveMsgx” is called automatically by the interrupt service routine. However some additions are required to be added in order to save the buffered data into an array.

CAN_1_TX_RX_func.c

```
#include "CAN_1.h"
/* `#START TX_RX_FUNCTION` */

#define DATALENGTH1 0x08
extern uint8 TxMessage1[8];
extern uint8 RxMessage1[8];
extern uint8 RXDLC1;
extern uint8 RxFlag1;

uint8 bIndex;
.
.
.
.
```

Add this

```
uint8 CAN_1_SendMsg0(void)
    {
        uint8 result = CYRET_SUCCESS;

        if ((CAN_1_TX[0u].txcmd.byte[0u] & CAN_1_TX_REQUEST_PENDING) ==
            CAN_1_TX_REQUEST_PENDING)
        {
            result = CAN_1_FAIL;
        }
        else
        {
            /* `#START MESSAGE TxDATA1 TRASMITTED` */
            for (bIndex=0;bIndex<DATALENGTH1;bIndex++)
            {
                  CAN_1_TX_DATA_BYTE(0,bIndex) = TxMessage1[bIndex];
            }
            /* `#END` */

            CY_SET_REG32((reg32 *) &CAN_1_TX[0u].txcmd,
CAN_1_SEND_MESSAGE);
        }

        return(result);
       }
.
.
.
.
void CAN_1_ReceiveMsg0(void)
    {
        /* `#START MESSAGE 0 RECEIVED` */
            RxFlag1 = 1; //Set flag
            RXDLC1 = CAN_1_GET_DLC(0); // Get no. of received bytes
            for (bIndex=0;bIndex<RXDLC1;bIndex++
            {
                  RxMessage1[bIndex] = CAN_1_RX_DATA_BYTE(0,bIndex);
            }

        /* `#END` */

            CAN_1_RX[0u].rxcmd.byte[0u] |= CAN_1_RX_ACK_MSG;

    }
```

Add this

Add this

3- The following C code is the main code

<u>main.c</u>

```
#include <device.h>
#include "stdio.h"

uint8 TxMessage1[8];
uint8 RxMessage1[8];
uint8 RXDLC1;
uint8 RxFlag1;


#define MAX_SAMPLE                  16

void main(void)
{
```

```
    /* Variable to hold ADC count */
    int32 voltCount = 0;

    /*Variable to hold the result in millivolts converted from ADC counts*/
    int32 mVolts = 0;

    /* Variable to count number of samples collected from ADC */
    uint8 sampleCount = 0;

    /* Variable to hold cumulative samples */
    int32 voltSamples = 0;

    /* Variable to hold the average volts for 64 samples */
    uint32 averageVolts = 0;

    /* Character array to hold the micro volts*/
    char displayStr[6] = {'\0'};

    CYGlobalIntEnable;

    /* Initiate and start CAN */
    CAN_1_Init();
    CAN_1_Start();

    /* Start ADC and start conversion */
    ADC_Start();
    ADC_StartConvert();

    /* Start LCD and set position */
    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintString("ADC input");

      LCD_Position(1,0);
      LCD_PrintString("CAN:");

    while(1)
    {
        /* Read ADC count and convert to milli volts */
        ADC_IsEndConversion(ADC_WAIT_FOR_RESULT);
        voltCount = ADC_GetResult16();
        mVolts = ADC_CountsTo_mVolts(voltCount);

        /* Add the current ADC reading to the cumulated samples*/
        voltSamples = voltSamples + mVolts;

        sampleCount++;

        /* If 16 samples have been collected then average the samples
           and update the display*/
        if(sampleCount == MAX_SAMPLE)
        {
            averageVolts = voltSamples >> 4;
            voltSamples = 0;
            sampleCount = 0;

            /* Convert milli volts to string and display on the LCD.
            *  sprintf() function is standard library function defined in
               the stdio.h header file */
            sprintf(displayStr,"%4ldmV",averageVolts);
```

```
            LCD_Position(0,10);
            LCD_PrintString(displayStr);

            // Save the 32 bit value in the CAN transmitting array
            TxMessage1[0]=displayStr[0];
            TxMessage1[1]=displayStr[1];
            TxMessage1[2]=displayStr[2];
            TxMessage1[3]=displayStr[3];
            CAN_1_SendMsg0();        // Transmit from first Tx buffer
        }
            ///////////////////////////////////
        if(RxFlag1)
            {
        /* Print the four received byte on an array, then display the array
         on the LCD  */
                  sprintf(displayStr,"%c%c%c%cmV",RxMessage1[0],RxMessage1[
                  1],RxMessage1[2],RxMessage1[3]);
                  LCD_Position(1,4);
                  LCD_PrintString(displayStr);
                  RxFlag1=0; // Reset flag
            }
    }
}

/* [] END OF FILE */
```

5. Software Setup – Node 2

In node 2 (refer to **Figure 5**) a third party controller is used. Node 2 works as repeater, it receives the message on ID 0x11111111 and then send the same message on ID 0x200. Flow chart is shown in Figure 14 below.

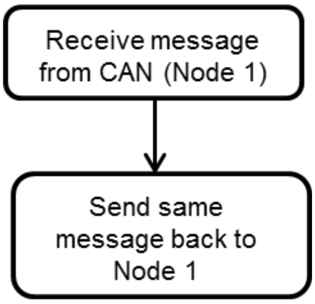

Figure 14: Flowchart- Node 2

6. Bugs

It is observed that sometimes the modifications in CAN_1_TX_RX_func.c are not saved after building the project, so check the file after building the project.

7. Downloads

- Project files can be downloaded from http://ia600609.us.archive.org/2/items/CanBusInPsoc5_522/voltageDisplay_SAR_ADC.zip
- Video demonstration can be watched on http://www.youtube.com/watch?v=Wn-wYuFVEJE

8. References:

1- CAN Bus, http://www.interfacebus.com/Design_Connector_CAN.html (retrieved on 12.01.2011).
2- Industrial CAN transceiver, SN65HVD251P data sheet, http://focus.ti.com/lit/ds/symlink/sn65hvd251.pdf (retrieved on 14.01.2011).
3- CAN Controller V1.50 data sheet, PSoC Creator 1.0 Beta 5.0.
4- Implementing CAN Bus Communication using PSoC® 3 and PSoC 5 - AN52701, http://www.cypress.com/?rID=37766 (retrieved on 14.01.2011).
5- CAN in 30 minutes or less, http://www.cypress.com/?docID=19093 (retrieved on 14.01.2011)